Architectures of Enterprise Systems: Modelling Transactional Contexts

Iman Poernomo[1], Ralf Reussner[1], and Heinz Schmidt[2]

[1] DSTC Pty Ltd, Monash University, Caulfield East, Victoria, Australia 3145,
{imanp,reussner}@dstc.com
[2] School of Computer Science and Software Engineering, Monash University, Caulfield East, Victoria, Australia 3145,
hws@csse.monash.edu.au

Abstract. Software architectural description languages (ADLs) are used to specify a high-level, compositional view of a software application, defining how a system is to be composed from coarse-grain components. ADLs usually come equipped with a rigourous state-transition style semantics, enabling formal understanding of distributed and event-based systems [6]. However, additional expressive power is required for the description and understanding of enterprise-scale software architectures – in particular, those built upon newer middleware, such as implementations of Java's EJB specification [2] or Microsoft's COM+/.NET [8]. Such middleware provides additional functionality to a configuration of components, by means of a context-based interception model [12]. We explore an ADL that can define architectures built upon such middleware. In this paper, we focus on modelling transactional architectures built on COM+ middleware.

1 Introduction

Over the past decade, middleware has undergone considerable evolution to meet the needs of the enterprise. The enterprise requires software solutions which are business-oriented, mission-critical, maintainable, flexible and distributed. It is now common to deliver such solutions by utilizing a component-based middleware, such as Java's EJB specification [2] or Microsoft's COM+/.NET [8]. A key feature of such middleware is the provision of infrastructure to support integration of cross-component functionality at the configuration level. For example, the COM+ architect may specify security or transactional settings over groups of interoperating components through Component Services. Such specification results in the addition of pre-programmed functionality to components, enabling the developer to focus on business-oriented design and programming.

In COM+, this kind of cross-component functionality is provided through context-based model of call interception. Here, deployed components are conceived as residing within a context that potentially intercepts and manipulates each call that crosses the context boundary. The advantage of this approach is that the middleware provides the ability for contexts to manage the interaction

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 233–243, 2002.

between components, without the need for the programmer to write management code. In EJB implementations, a similar result is achieved through the container-server model. The ubiquity of this kind of computation has resulted in the need for the enterprise system architect to design systems that involve both components and contexts. In this paper, we outline an approach to modelling architectures that use contexts. Here, we consider software architecture to be a compositional view of how coarse-grain system elements (such as COM+ components and contexts) are assembled to form a piece of software. We extend architectural description language (ADL) approaches to meet our goal [6,5].

We focus on transactional contexts. These contexts add transactional functionality to components, without the need for the programmer to write transactional code. Transactional contexts make the job of the programmer easier, eliminating the need to write management code. However, without the use of a formal architectural approach to contexts, the architect runs the risk of error. Currently, many designers use an informal architectural style to define context-component relationships in a system. Our work is novel, as it presents the first formal architectural description language that includes such relationships in its system descriptions. While our approach focuses on modelling COM+ transactional contexts, a similar approach can be taken for EJB implementations.

Our approach is as follows. In Section 2, we define an architectural description language (ADL) for middleware-based architectures in terms of components and contexts. In Section 3, we give a semantics for our architectural descriptions, modelling component and transactional context use via finite state machines.

2 Architectural Description

The TrustME ADL is fully described in [9]. Like other ADLs, it provides a means of defining compositions of component-based systems [5,6]. In this paper, we outline the subset of TrustME relevant to describing transactional contexts for COM+ middleware-based architectures, the grammar of which is given in Fig. 1.

In modelling middleware-based systems, we decompose a system into hierarchies of contexts, each containing components, linked to each other by connections between their services:

- *Components* are self-contained, coarse-grain computational entities, potentially hierarchically composed from other components. Our ADL's components are intended to directly represent COM+ components of an implementation.
- *Services* represent an abstraction of the type of messages that may be sent between components. A service may be either provided or required by a component.
- *Contexts* are used to model transactional contexts of COM+ (and can be extended to model container/servers of EJB implmentations).

Components and services are analogous to components and interfaces in Darwin, to components and ports respectively in C2 and ACME, or to processes and ports

Grammar for component declarations, $ComDec$

```
component   ComType    {
            (Comp : ComType)*
            (bind(Comp.Service, Comp.Service))*
            map(Comp.Service, Service)
            provides : (Service)*   requires : (Service)*    }
```

Grammar for transactional context declarations, $ContextDec$

```
context   ContextType    {
          ((Context : ContextType)|(Comp : ComType))*
          bind(Context|Comp.Service, Context|Comp.Service)
```

Grammar for architectures

```
architecture        {
                    (ComDec|ContextDec)*
                    ((Context : ContextType)|(Comp : ComType))*
                    bind(Comp.Service, Comp.Service)*
                    map(Context|Comp.Service, Service)*          }
```

Fig. 1. The grammar for components, contexts and architectures. $ComType$, $ContextType$ and $Service$ range over a sets of names.

in MetaH [5,6]. However, our ADL differs from these other languages, in that it enables representation of transactional contexts as first class entities. This for modelling middleware-based systems.

Our ADL is a typed, class-based metamodel: contexts and component types are defined via class declarations, and may be instantiated to be reused within other contexts, components or architecture definitions. To instantiate a component C of type T, we write $C : T$. We define the subcomponents of a component or context $C : T$ to consists of all components contained in the declaration of T, together with their subcomponents. We write $subcomponents_C$ for this set. The set of subcontexts $subcontext_C$ of a context $C : T$ is similarly defined, consisting of all contexts contained in the declaration of T, together with their subcontexts.

A service s of a component A may be referred to within a larger component, context or an architecture by a C++/Java style qualification: $A.s$. Also, a provided service s of a component A that is defined within a context C may be referred to, outside the context, by a further qualification $C.A.s$. This reflects the fact that, in implementation, the provided services of a component within a transactional context may be called by a component outside of the context. Within a compound component, architecture or context, the required service of a subcomponent may use the provided service of another component. This is defined via a `bind` declaration. A compound component exposes provided services which may delegate calls to provided services subcomponents. The same may be said of required services. This is defined via a `map` declaration.

$Declare(HotelRes) \equiv$	`component` $CHotelRes\{$`provides : bookHotel`, $SQL\_op, abort, commit\}$
$Declare(FlightRes) \equiv$	`component` $CFlightRes\{$`provides : bookFlight`, $abort, commit\}$
$Declare(ResSystem) \equiv$	`component` $CResSystem\{$ `provides : processReservation` `requires : makeBilling, makeReservation`, $abort, commit\}$
$Declare(ResTransfer) \equiv$	`component` $CResTransfer\{Declare(HotelRes)$, $Declare(FlightRes), HotelRes : CHotelRes$, $FlightRes : CFlightRes$ `map`$(processReservation, HotelRes.bookHotel)$ `map`$(processReservation, FlightRes.bookFlight)$ `provides` : $makeReservation$, $abort, commit\}$
$Declare(ResBill) \equiv$	`component` $CResBill$ $\{$`provides` : $makeBilling$, $abort, commit\}$

Fig. 2. Five COM+ components, specified as ADL component declarations. We abbreviate these declarations by $Declare(CResSystem)$, $Declare(CResTransfer)$ and $Declare(CResBill)$ in the rest of the paper.

Example 1. Our example architecture involves a simple hotel reservation system, built from three COM+ components – $ResSystem$, $ResTransfer$, $ResBill$. These components are instances of class types $CResSystem$, $CResTransfer$, $CResBill$. The component $ResTransfer$ uses subcomponents $FlightRes$ (of type $CFlightRes$) and $HotelRes$ (of type $CHotelRes$). Upon recieving an event notification from the first component that a hotel reservation is to be made by a user, the second component performs B2B operations with the Hotel and the airliner for which the reservation is made. Upon receiving the same type of event notification, the third component performs billing operations against the user's credit card. When the event is sent out, $ResTransfer$ and $ResBill$ will execute concurrently. However, we require transaction support over both these components: if one fails, then calls to either component must be rolled back. This will prevent a user being billed if the hotel they wish to book at is full, and will prevent the hotel from accepting a guest if their credit is bad. We require separate transaction support for calls to $FlightRes$ and $HotelRes$ from within $ResTransfer$, to prevent a flight being reserved if the hotel is full. Fig. 2 defines the components of our example in the syntax of TrustME.

2.1 Architectures

In implementation, the designer defines transactional contexts by assigning transaction settings to component deployments. We reflect this by considering transactional contexts as first class entities in our ADL.

There are five possible transaction settings in COM+ (there are similar settings for EJB implementations). *1. Disabled.* This setting means that a component is not transactional, and therefore should not make calls to any managed resources, or to componenents which managed resources. *2. Not supported.* This setting indicates that the component will not run within a transaction. If the component is called from within a transaction's activity, COM+ creates the component in a context running outside of the current activity. *3. Supported.* This indicates that the component will run in a transaction, but does not require one. *4. Required.* This indicates that the component requires a transaction to run. If the component is not currently in a transaction, COM+ will start a new transaction. *5. Requires new.* This indicates that the component requires a new transaction. If the component is called from within a transaction, COM+ will start a new transaction activity for this component. Use of this setting ensures that this component's transaction is unaffected by the success or failure of others.

A transactional COM+ architecture is implemented by assigning these settings to a group of COM+ components. We model such implementations by means of contexts.

Example 2. Our architecture will consist of all components contained within the same transactional context, and with $ResBill$ (and thus its subcomponents) contained within a subcontext. Our architecture is depicted in Fig. 3. The required services of $ResSystem$, $makeReservation$ and $makeBilling$, are bound to the provided services $makeReservation$ and $makeBilling$ of $ResTransfer$ and $ResBill$ respectively. This specifies that when $ResSystem$ requests that the reservation be made and billed, $ResTransfer$ and and $ResBill$ will respectively handle these functions. To meet our transactional requirements, the three COM+ components $ResSystem$, $ResTransfer$ and $ResBill$ have settings of *required, supported* and *supported* respectively. A transactional context intercepts calls made to and from $ResBill$. This is represented by declaring a context type $CResBillContext$ and context instance $ResBillContext$ that uses $ResBill$. The the required services of other components are bound to the provided services of $ResBill$ inside this context.

```
architecture {
 Declare(CResSystem), Declare(CResTransfer), Declare(CResBill)
 context CResSystemContext {
  ResSystem : CResSystem, ResTransfer : CResTransfer, ResBill : CResBill
  bind(ResSystem.makeReservation, ResTransfer.makeReservation),
  bind(ResSystem.makeBilling, ResBill.makeBilling)                    }
 ResSysContext : CResSystem, ResTransfer : CResTransfer, ResBill : CResBill
 bind(ResSystem.makeReservation, ResTransfer.makeReservation),
 bind(ResSystem.makeBilling, ResBillContext.makeBilling)              }
```

Fig. 3. An architecture in which the three components are in a transactional context, with $ResTransfer$ (and its subcomponents) in a separate, nested transaction.

Note that non-transactional systems can still be defined, using components and connections without contexts. The form of these architectures are similar those that are definable in other ADLs, involving a collection of components, together with binding connections between their provided and required services.

3 Semantics

The language of our ADL depicts the static structure of a system. To understand the dynamic behaviour of a system, we define a semantics for the elements of our language.

3.1 Semantics of Basic Components

We do not try to give a comprehensive description of a component's sematics. Since components themselves are able to perform arbitrary universal computations, a comprehensive description of their semantics also would require a model of universal computational power. Unfortunately, universal models are hard to analyse and many interesting properties are undecidable.

Hence, we restrict our semantics to describe the aspect of input-output behaviour. We model this aspect with finite-state machines (FSMs), for which efficient algorithms exist for testing equivalence and inclusion and performing liveness and safety analysis.

Definition 1 (Finite State Machine). *A Deterministic Finite State Machine (FSM) D is a tuple $D = (E, A, Z, \delta, F, z_0)$. E is called event alphabet, A is the event alphabet, Z denotes the set of states, $F \subseteq Z$ is the set of accepting (final) states. $z_0 \in Z$ is a designated* start-state. *$\delta : Z \times E \times A \to Z$ is a* total *function.*

We often denote the transition function as a set of transitions, where we write each transition as a tuple $(fromState, event, action, toState)$, meaning that $\delta(fromState, event, action) = toState$ and that the action *action* is associated with that state transition.

A basic component has a semantics given by a FSM, whose event-alphabet consists of the component's provided services and whose action-alphabet consists out of the component's required services. Our semantics assumes that every basic component $C : T$ has a so called *component-FSM* (CFSM) describing its input-output behaviour.

Definition 2 (Component-FSM). *For a given component C, the component-FSM (CFSM) C-FSM$_C = (E_C, A_C, Z_C, \delta_C, F_C, z_{0_C})$ is defined as follows:*

- *for each service $m \in provides_C \cup internal_C$ two distinct events m_{start} and m_{end} exist in the event alphabet E_C. m_{start} denotes the call of service m, m_{end} its return.*
- *the action alphabet A_C consists of the component's required services.*

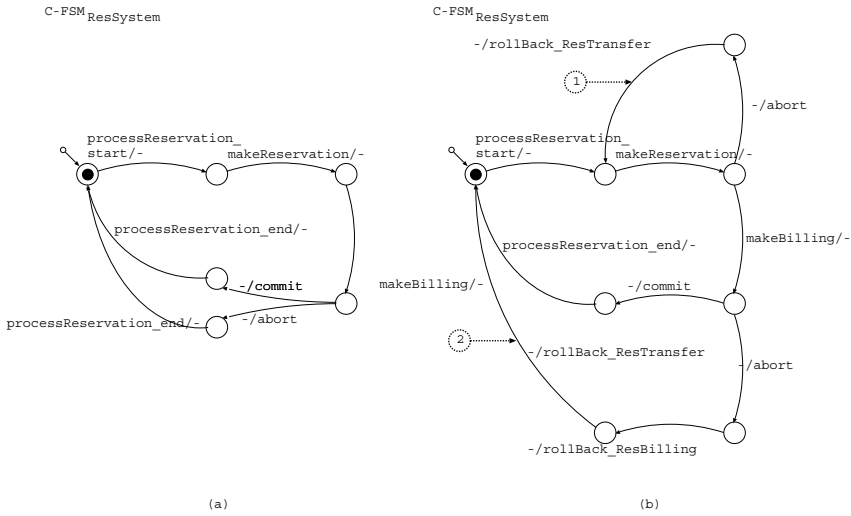

Fig. 4. (a): C-FSM of the *ResSystem* component. (b): Transactional C-FSM of the *ResSystem* component.

The C-FSM of a component provides a mapping from called provided services to resulting call made to external components via required services. The transition function δ for a component defines the possible changes to the the state of the component that occur when it receives events through its provided services, or when it sends events through its required services. Each transition is annotated with either an event or an action.

Example 3. In Fig. 4(a) we present the C-FSM of the *ResSystem* component.

3.2 Semantics of Compound Components and Nontransactional Architectures

The semantics of a compound component is given by a C-FSM, which is derived from the semantics of the compound component's subcomponents. The idea is to examine the bindings between provided and required services of the components and provided and required services of subcomponents respectively. This information can be used to define FSMs that model an entire sequence of actions and events that result from calling provided services. We refer to these FSMs as service FSMs. Then, we examine the mappings from the provided and required services of C to provided and required services of subcomponents. We combine this information with service FSMs to model the event and action sequences that can result from calling the component's provided services. This results in the component's C-FSM. See [11,10] for details.

An architecture that consists solely of hierarchically composed components (without transactional contexts) has a semantics defined in a similar way to that

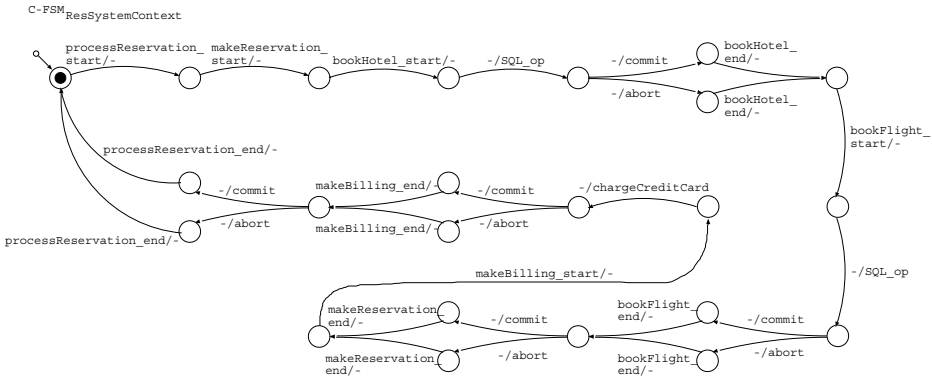

Fig. 5. C-FSM of the nontransactional architecture obtained by removing transactional contexts from Fig. 3.

of compound components. As an example one can consider Fig. 5, the C-FSM for the architecture described in Fig. 3 with transactional contexts removed.

3.3 Architectures with Transactional Contexts

Transactional contexts change the way contained components interoperate. This is mirrored through changes to the overall behavioural semantics for an architecture. We model transactional contexts. Just as a middleware can add transactional cross-component functionality, the presence of contexts in an architecture adds additional behaviour to the semantics of contained components.

Definition 3. *For a given FSM* $A = (E_A, A_A, Z_A, \delta_A, F_A, z_{0_A})$ *and component* C*, the mapping* $\Phi : Z_A \to \mathbf{P}(subcomponents_C)$ *assigns for each state* $s \in Z_A$ *a set of components* $D \subseteq Z_A$*, where* $d \in D$ *is true if (and only if) on a path from the start state* z_{0_A} *to the state* s *a service* p *with* $\zeta(p) = d$ *has been called.*

Definition 4 (Rollback-Automata). *For a given set of states* $D \subseteq Z_A, D = \{d_1 \cdots d_n\}$ *of a given FSM* A *we define the* rollback-automata *as a FSM* $r_A(D) = (E_r, A_r, Z_r, \delta_r, F_r, z_{0_r})$*, where* $Z_r{:=}l_0 \cdots l_n$*,* $z_{0_r}{:=}l_0$*,* $F_r{:=}\{l_n\}$ *and* $\delta_r(l_{i-1}, d_i){:=} l_i$*, for* $i \in [1..n]$

Algorithm 1 (Translation of CR-FSM into transactional CR-FSM)

Input: non-transactional CR-FSM $A = (E_A, A_A, Z_A, \delta_A, F_A, z_{0_A})$
Output: transactional CR-FSM $B = (E_{tr(A)}, A_{tr(A)}, Z_{tr(A)}, \delta_{tr(A)}, F_{tr(A)}, z_{0_{tr(A)}})$

Build mapping $\Phi : Z \to \mathbf{P}(subcomponents_C)$;
⟨e.g., by dataflow algorithms.⟩
$B{=}A$.clone();

for each set $s \in dom(\Phi)$ **do**
add rollback-automata $r_A(s)$ to B;
add transition $(\Phi^{-1}(s), abort, z_{0_r})$ to B;
state $f := l.l \in F_r$;
identify states z_{0_A} and f;
⟨do not make z_{0_A} a final state, unless it allready has been one.⟩
od

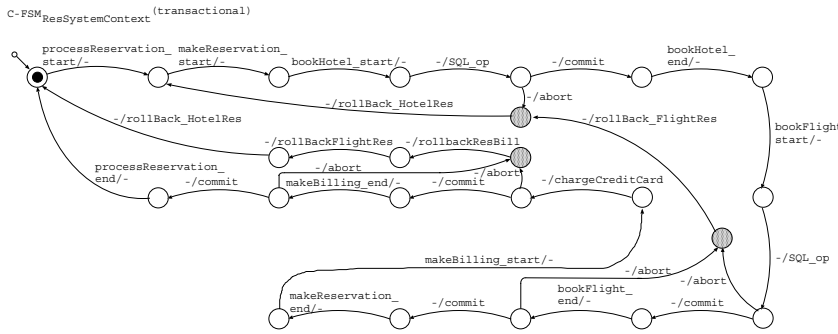

Fig. 6. Transactional C-FSM for the architecture of Fig. 3.

Example 4. Applying this algorithm to the non-transactional C-FSM of *ResSystem* results in the transactional C-FSM given in Fig. 4(b).

Nested transactional contexts can be treated by applying algorithm 1 to the innermost transactional contexts, then recursively to outer contexts. Thus, the C-FSM of the overall context has to be constructed. The sub-FSMs of the inner transactional contexts can be identified by the start states of their services. The part of the overall C-FSM which already has been processed as inner contexts, must be marked and excluded from further processing of the outer contexts. In the example of figure 4(b), the rollback-transition marked as 1 leads to the beginning of the inner transactional context for the component $ResTransfer$ because its triggering *abort*-transition occurs within the inner transactional context. The rollback-transition marked as 2 leads to the outer transactional context, since it results from an *abort* transition of the outer transactional context.

In Fig. 6 we show the result of applying algorithm 1 on the C-FSM of the the architecture of Fig. 3. Since some rollback automata are combined, we marked the start states of rollback automata as shaded states.

4 Related Work and Conclusions

In this paper we examine extensions to ADLs that facilitate modelling of architectures involving transactional contexts. Our main contributions are: 1. We

define a simple ADL for textually defining industrial middleware architectures, 2. we provide a compositional finite state machine based semantics for compositional component architectures, 3. we extend this semantics to incorporate architectures that involve transactional contexts.

Our basic approach to syntax and semantics is similar to that of other ADLs . Some form of semantics is used to define models of basic components' behaviour. For example, Darwin [5] uses the π-calculus [7], Rapide [4] uses partially order event sets and Wright [1] can use a form of CSP [3]. The language of the ADL is used primarily to impose structure on over a system's behavior, expressing its composition from components. Semantically, the composition-forming constructs are associated with semantic functions, which describe how a composite component's behavioural semantics is formed from subcomponents' behavioural semantics.

However, to the best of our knowledge, no work has been done previously on the syntax and semantics of architectures of middleware-based systems with transactional contexts. The advantage of finite state machines when modelling component-architectures is the availability of efficient checks for interoperability and substitutability. However, further work needs to be done to adapt our approach to more expressive forms of semantics, such as the π-calculus. Also, here we have only modelled transactional contexts: we are currently extending our system to model other middleware contexts.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful suggestions and Wolfgang Blass for his fruitful help.

References

1. Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PE, USA, May 1997.
2. Sun Microsystems Corp., The Enterprise Java Beans homepage. http://java.sun.com/products/ejb/.
3. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice/Hall, 1985.
4. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr 1995.
5. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989:137–155, 1995.
6. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Janurary 2000.
7. Robin Milner. Concurrency and compositionality, 1991. Workshop at Goslar.
8. Microsoft Corp., The .net homepage. http://www.microsoft.com/net/default.asp.
9. Iman Poernomo, Heinz Schmidt, and Ralf Reussner. The TrustME language site. Web site, DSTC, 2001. Available at `http://www.csse.monash.edu.au/dsse/trustme`.

10. Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.
11. Heinz W. Schmidt and Ralf H. Reussner. Automatic Component Adaptation By Concurrent State Machine Retrofitting. Technical Report 25/2000, Fakultät für Informatik, Universität Karlsruhe (TH), Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 2000.
12. Clemens Szyperski. Components and architecture. *Software Development*, 8(5), October 2000.